

Connector 2.0

- User Guide -

C. Benoit, G. Jeanfaivre, S. Peron, P. Raud, S. Landier, B. Rodriguez, S. Verley
Onera / DSNA

1 Connector: Grid Connectivity module

1.1 Preamble

Connector module is used to compute connectivity between meshes. It manipulates arrays and pyTree as data structures. To use the Connector module with array interface:

```
import Connector as X
```

and with pyTree interface:

```
import Connector.PyTree as X
```

1.2 Connectivity with arrays

X.connectMatch: detect and set all matching windows, even partially between two structured arrays a1 and a2. Returns the subrange of indices of abutting windows and an index transformation from a1 to a2. If the CFD problem is 2D, then dim must be set to 2. Parameter sameZone must be set to 1 if a1 and a2 define the same zone:

```
res = X.connectMatch(a1, a2, sameZone=0, tol=1.e-6, dim=3)
```

(See: Examples/Connector/connectMatch.py)

X.blankCells: blank the cells of a list of grids defined by coords (located at nodes). The X-Ray mask is defined by bodies, which is a list of arrays. Cellnaturefield defined in cellns is modified (0: blanked points, 1: otherwise).

Some parameters can be specified: blankingType, delta, masknot, tol. Their meanings are described in the table below:

Parameter value	Meaning	
blankingType=0	blank nodes inside bodies (node_in).	
blankingType=2	blank cell centers inside bodies (center_in).	
blankingType=1	blank cell centers intersecting with body (cell_intersect).	
blankingType=-2	blank cell centers using an optimized cell intersection (cell_intersect_opt) and interpolation depth=2 (blanking region may be reduced where blanking point can be interpolated).	
blankingType=-1	blank cell centers using an optimized cell intersection (cell_intersect_opt) and interpolation depth=1.	
delta=0.	cells are blanked in the body	
delta greater than 0.	the maximum distance to body, in which cells are blanked	
masknot=0	Classical blanking applied	
masknot=1	Inverted blanking applied: cells out of the body are blanked	
dim=3	body described by a surface and blanks 3D cells.	
dim=2	body blanks 2D or 1D zones.	
tol=1.e-8 (default)	tolerance for the multiple definition of the body.	

Warning: in case of blankingType=0, location of cellns and coords must be identical.

```
cellns = X.blankCells(coords, cellns, body, blankingType=2, delta=1.e-10, dim=3, masknot=0, tol=1.e-8)
```

(See: Examples/Connector/blankCells.py)

X.setHoleInterpolatedPoints: compute the fringe of interpolated points around a set of blanked points in a mesh a. Parameter depth is the number of layers of interpolated points to be set. If depth ≥ 0 the fringe of interpolated points is set outside the blanked zones, whereas if depth < 0 , the depth layers of blanked points are marked as to be interpolated. If dir=0, uses a directional stencil of depth points, if dir=1, uses a full depth x depth x depth stencil: Blanked points are identified by the variable 'cellN'; 'cellN' is set to 2 for the fringe of interpolated points:

```
a = X.setHoleInterpolatedPoints(a, depth=2, dir=0)
```

(See: Examples/Connector/setHoleInterpolatedPts.py)

X.optimizeOverlap*: Optimize the overlap between two zones defined by nodes1 and nodes2, centers1 and centers2 correspond to the mesh located at centers and the field 'cellN'. The field 'cellN' located at centers is set to 2 for interpolable points.

Priorities can be defined for zones: prio1=0 means that the priority of zone 1 is high. If two zones have the same priority, then the cell volume criterion is used to set the cellN to 2 for one of the overlapping cells, the other not being modified.

If the priorities are not specified, the cell volume criterion is applied also:

```
cellns = X.optimizeOverlap(nodes1, centers1, nodes2, centers2, prio1=0, prio2=0)
```

(See: Examples/Connector/optimizeOverlap.py)

X.maximizeBlankedCells: change useless interpolated points status (2) to blanked points (0). If dir=0, uses a directional stencil of depth points, if dir=1, uses a full depth x depth x depth stencil:

```
b = X.maximizeBlankedCells(a, depth=2, dir=0) .or. B = X.maximizeBlankedCells(A, depth=2)
```

(See: Examples/Connector/maximizeBlankedCells.py)

X.setDoublyDefinedBC: when a border of zone z is defined by doubly defined BC in range=[i1,i2,j1,j2,k one can determine whether a point is interpolated or defined by the physical BC. The array cellN defines the cell nature field at centers for zone z. If a cell is interpolable from a donor zone, then the cellN is set to 2 for this cell. The lists listOfInterpZones and listOfCelln are the list of arrays defining the interpolation domains, and corresponding cell nature fields. depth can be 1 or 2. If case of depth=2, if one point of the two layers is not interpolable, then celln is set to 1 for both points:

```
t = X.setDoublyDefinedBC(z, cellN, listOfInterpZones, listOfCelln, range, depth=2)
```

(See: Examples/Connector/setDoublyDefinedBC.py)

X.blankIntersectingCells: blank intersecting cells of a 3D mesh. Only faces normal to k-planes for structured meshes and faces normal to triangular faces for prismatic meshes, and faces normal to 1234 and 5678 faces for hexahedral meshes are tested. The cellN is set to 0 for intersecting cells/elements. Input data are A the list of meshes, cellN the list of cellNatureField located at cell centers. Array version: the cellN must be an array located at centers, defined separately:

```
cellN = X.blankIntersectingCells(A, cellN, tol=1.e-10)
```

(See: Examples/Connector/blankIntersectingCells.py)

1.3 Connectivity with pyTrees

X.setDegeneratedBC: detect all degenerate lines in 3D zones and define a BC as a 'BCDegenerateLine' BC type. For 2D zones, 'BCDegeneratePoint' type is defined. If the problem is 2D according to (i,j), then parameter 'dim' must be set to 2. Parameter 'tol' defines a distance below which a window is assumed degenerated.

```
t = X.setDegeneratedBC(t,dim=3,tol=1.e-10)
```

(See: Examples/Connector/setDegeneratedBCPT.py)

X.connectMatch: detect and set all matching windows, even partially, in a zone node, a list of zone nodes or a complete pyTree. Set automatically the Transform node corresponding to the transformation from matching block 1 to block 2. If the CFD problem is 2D, then dim must be set to 2:

```
t = X.connectMatch(t, tol=1.e-6, dim=3)
```

(See: Examples/Connector/connectMatchPT.py)

X.connectMatchPeriodic: detect and set all periodic matching windows, even partially, in a zone node, a list of zone nodes, a basis, or a complete pyTree. Periodicity can be defined by a

rotation or by a translation or by a composition of rotation and translation. If the mesh is periodic in rotation and in translation separately (i.e. connecting with some blocks in rotation, and some other blocks in translation), the function must be applied twice. Set automatically the Transform node corresponding to the transformation from matching block 1 to block 2, and the 'GridConnectivityProperty/Periodic' for periodic matching BCs. If the CFD problem is 2D, then dim must be set to 2:

```
t = X.connectMatchPeriodic(t, rotationCenter=[0.,0.,0.], rotationAngle=[0.,0.,0.], translation=[0.,0.,0.], tol=1.e-6, dim=3)
```

(See: Examples/Connector/connectMatchPeriodicPT.py)

X.connectNearMatch: detect and set all near-matching windows, even partially in a zone node, a list of zone nodes or a complete pyTree. A 'UserDefinedData' node is set, with the PointRangeDonor, the Transform and NMRatio nodes providing information for the opposite zone. Warning: connectMatch must be applied first if matching windows exist. Parameter ratio defines the ratio between near-matching windows and can be an integer (e.g. 2) or a list of 3 integers (e.g. [1,2,1]), specifying the nearmatching direction to test (less CPU-consuming). If the CFD problem is 2D, then dim must be set to 2:

```
t = X.connectNearMatch(t, ratio=2, tol=1.e-6, dim=3)
```

(See: Examples/Connector/connectNearMatchPT.py)

X.getCEBBIntersectingDomains: detect the domains defined in the list of bases B whose CEBB intersect domains defined in base A. Return the list of zone names for each basis. If sameBase=1, the intersecting domains are also searched in base:

```
domNames = X.getCEBBIntersectingDomains(A, B, sameBase)
```

(See: Examples/Connector/getCEBBIntersectingDomainsPT.py)

X.getCEBBTimeIntersectingDomains: in a Chimera pre-processing for bodies in relative motion, it can be useful to determine intersecting domains at any iteration.

niter defines the number of iterations on which CEBB intersections are detected, starting from iteration inititer.

dt defines the timestep.

func defines a python function defining the motion of base, funcs is the list of python functions describing motions for bases.

Warning: 1. motions here are only relative motions. If all bases are translated with the same translation motion, it must not be defined in func.

2. If no motion is defined on a basis, then the corresponding function must be []:

```
domNames = X.getCEBBTimeIntersectingDomains(base, func, bases, funcs, inititer=0, niter=1, dt=1, sameBase)
```

(See: Examples/Connector/getCEBBTimeIntersectingDomainsPT.py)

X.applyBCOverlaps: set the cellN to 2 for the fringe nodes or cells (depending on parameter 'loc'='nodes' or 'centers') near the overlap borders defined in the pyTree t. Parameter 'depth' defines the number of layers of interpolated points.

```
t = X.applyBCOverlaps(t, depth=2, loc='centers')
```

(See: Examples/Connector/applyBCOverlapsPT.py)

X.blankCells: blankCells function sets the cellN to 0 to blanked nodes or cell centers of both structured and unstructured grids. The location of the cellN field depends on the blankingType pa-

parameter: if 'node_in' is used, nodes are blanked, else centers are blanked. The mesh to be blanked is defined by a pyTree t, where each basis defines a Chimera component. The list of bodies blanking the grids is defined in bodies. Each element of the list bodies is a set of CGNS/Python zones defining a closed and watertight surface.

The blanking matrix BM is a numpy array of size nbases x nbodies.

BM(i,j)=1 means that ith basis is blanked by jth body.

BM(i,j)=0 means no blanking, and BM(i,j)=-1 means that inverted hole-cutting is performed.

blankingType can be 'cell_intersect', 'cell_intersect_opt', 'center_in' or 'node_in'. Parameter depth is only meaningful for 'cell_intersect_opt'.

XRaydim1 and XRaydim2 are the dimensions of the X-Ray hole-cutting in the x and y directions in 3D.

If the variable 'cellN' does not exist in the input pyTree, it is initialized to 1, located at 'nodes' if 'node_in' is set, and at centers in other cases.

Warning: 'cell_intersect_opt' can be CPU time-consuming when delta_i0.

```
B = X.blankCells(t, bodies, BM, depth=2, blankingType='cell_intersect', delta=1.e-10, dim=3,
tol=1.e-8, XRaydim1=1000, XRaydim2=1000)
```

(See: Examples/Connector/blankCellsPT.py)

X.setHoleInterpolatedPoints: compute the fringe of interpolated points around a set of blanked points in a pyTree t. Parameter depth is the number of layers of interpolated points that are built; if depth ≤ 0 the fringe of interpolated points is outside the blanked zones, and if depth > 0 , it is built towards the inside. Blanked points are identified by the variable 'cellN' located at mesh nodes or centers. 'cellN' is set to 2 for the fringe of interpolated points.

```
t = X.setHoleInterpolatedPoints(t, depth=2)
```

(See: Examples/Connector/setHoleInterpolatedPtsPT.py)

X.optimizeOverlap: optimize the overlapping between all structured zones defined in a pyTree t. The 'cellN' variable located at cell centers is modified, such that cellN=2 for a cell interpolable from another zone.

Double wall projection technique is activated if 'double_wall'=1. The overlapping is optimized between zones from separated bases, and is based on a priority to the cell of smallest size. One can impose a priority to a base over another base, using the list priorities. For instance, priorities = ['baseName1',0, 'baseName2',1] means that zones from base of name 'baseName1' are preferred over zones from base of name 'baseName2':

```
t = X.optimizeOverlap(t, double_wall=0, priorities=[])
```

(See: Examples/Connector/optimizeOverlapPT.py)

X.maximizeBlankedCells: change useless interpolated cells (cellN=2) status to blanked points (cellN=0). Parameter depth specifies the number of layers of interpolated cells to be kept:

```
t = X.maximizeBlankedCells(t, depth=2)
```

(See: Examples/Connector/maximizeBlankedCellsPT.py)

X.cellN2OversetHoles: compute the OversetHoles node into a pyTree from the cellN field, located at nodes or centers. For structured zones, defines it as a list of ijk indices, located at nodes or centers. For unstructured zones, defines the OversetHoles node as a list of indices ind, defining the cell vertices that are of cellN=0 if the cellN is located at nodes, and defining the cell centers that are of cellN=0 if the cellN is located at centers. The OversetHoles nodes can be then dumped

to files, defined by the indices of blanked nodes or cells:

```
t = X.cellN2OversetHoles(t)
```

(See: Examples/Connector/cellN2OversetHolesPT.py) (See: Examples/Connector/oversetHoles2File.py)

X.setDoublyDefinedBC: when a border is defined by doubly defined BC, one can determine whether a point is interpolated or defined by the physical BC. The cellN is set to 2 if cells near the doubly defined BC are interpolable from a specified donor zone:

```
t = X.setDoublyDefinedBC(t, depth=2)
```

(See: Examples/Connector/setDoublyDefinedBCPT.py)

X.blankIntersectingCells: blank intersecting cells of a 3D mesh. Only faces normal to k-planes for structured meshes and faces normal to triangular faces for prismatic meshes, and faces normal to 1234 and 5678 faces for hexahedral meshes are tested. Set the cellN to 0 for intersecting cells/elements. Input data are A the list of meshes, cellN the list of cellNatureField located at cell centers: The cellN variable is defined as a FlowSolution#Center node. The cellN is set to 0 for intersecting and negative volume cells:

```
a = X.blankIntersectingCells(a, tol=1.e-10, depth=2)
```

(See: Examples/Connector/blankIntersectingCellsPT.py)

X.setInterpData*: set interpolation data (donor and receiver points, interpolation type, interpolation coefficients) given receivers defined by aR, donor zones given by aD. If storage='direct', then aR with interpolation data stored in receiver zones are returned, and if storage='inverse', then aD with interpolation data stored in donor zones are returned. Donor zones can be structured or unstructured TETRA. Receiver zones can be structured or unstructured.

Interpolation order can be 2, 3 or 5 for structured donor zones, only order=2 for unstructured donor zones is performed.

Parameter loc can 'nodes' or 'centers', meaning that receiver points are zone nodes or centers.

penalty=1 means that a candidate donor cell located at a zone border is penalized against interior candidate cell.

nature=0 means that a candidate donor cell containing a blanked point(cellN=0) is not valid. If nature=1 all the nodes of the candidate donor cell must be cellN=1 to be valid.

double_wall=1 activates the double wall correction. If there are walls defined by families in aR or aD, the corresponding top trees topTreeRcv or/and topTreeDnr must be defined.

If sameName=1, interpolation from donor zones with the same name as receiver zones are avoided.

Interpolation data are stored as a ZoneSubRegion_t node, stored under the donor or receiver zone node depending of the storage:

```
a = X.setInterpData(aR, aD, double_wall=0, order=2, penalty=1, nature=0, loc='nodes', storage='direct', topTreeRcv=None, topTreeDnr=None, sameName=0)
```

(See: Examples/Connector/setInterpDataPT.py)

X.setInterpTransfers: General transfers from a set of donor zones defined by topTreeD to receptor zones defined in aR.

Both Chimera and IBC transfers can be applied and are identified by the prefix name of the ZoneSubRegion node created when computing the overset or IBC interpolation data.

Parameter variables is the list of variable names that are transferred by Chimera interpolation.

Parameter variablesIBC defines the name of the 5 variables used for IBC transfers.

Parameter bcType can be 0 or 1 (see table below for details).

Parameter varType enables to define the meaning of variables IBC, if their name is not standard (see table below for more details).

Parameter storage enables to define how the information is stored (see table below).

Parameter	Meaning	
bcType = 0	IBC transfers model slip conditions	
bcType = 1	IBC transfers model no-slip conditions	
varType = 1	Density, MomentumX, MomentumY, MomentumZ, EnergyStagnationDensity	
varType = 2	Density, VelocityX, VelocityY, VelocityZ, Temperature	
varType = 3	Density, VelocityX, VelocityY, VelocityZ, Pressure	
storage = 0	interpolation data is stored in receptor zones aR	
storage = 1	interpolation data is stored in donor zones topTreeD	
storage = -1	interpolation data storage is unknown or can be stored in donor and receptor zones.	

```
aR = X.setInterpTransfers(aR, topTreeD, variables=None, variablesIBC=['Density', 'MomentumX', 'MomentumY', 'MomentumZ', 'EnergyStagnationDensity'], bcType=0, varType=1, storage='unknown')
```

(See: Examples/Connector/setInterpTransfersPT.py)

X.getOversetInfo: set information on Chimera connectivity, i.e. interpolated, extrapolated or orphan cells, donor aspect ratio and ratio between volume of donor and receiver cells. This function is compliant with the storage as defined for setInterpData function. If type='interpolated', variable 'interpolated' is created and is equal to 1 for interpolated and extrapolated points, 0 otherwise. If type='extrapolated', variable 'extrapolated' is created and its value is the sum of the absolute values of coefficients, 0 otherwise. If type='orphan', variable 'orphan' is created and is equal to 1 for orphan points, 0 otherwise. If type='cellRatio', variable 'cellRatio' is created and is equal to max(volD/volR, volR/volD) for interpolated and extrapolated points (volR and volD are volume of receivers and donors). If type='donorAspect', variable 'donorAspect' is created and is equal to the ratio between the maximum and minimum length of donors, and 0 for points that are not interpolated:

```
tR = X.getOversetInfo(tR, tD, type='interpolated')
```

(See: Examples/Connector/getOversetInfoPT.py)

X.setIBCData*: set IBC data (donor and receiver points, interpolation type, interpolation coefficients, coordinates of corrected, wall and interpolated points) given receivers defined by aR, donor zones given by aD. If storage='direct', then aR with interpolation data stored in receiver zones are returned, and if storage='inverse', then aD with interpolation data stored in donor zones are returned. Donor zones can be structured or unstructured TETRA. Receiver zones can be structured or unstructured.

Interpolation order can be 2, 3 or 5 for structured donor zones, only order=2 for unstructured donor zones is performed.

Parameter loc can 'nodes' or 'centers', meaning that receiver points are zone nodes or centers.

penalty=1 means that a candidate donor cell located at a zone border is penalized against interior candidate cell.

nature=0 means that a candidate donor cell containing a blanked point(cellN=0) is not valid. If nature=1 all the nodes of the candidate donor cell must be cellN=1 to be valid.

Interpolation data are stored as a ZoneSubRegion.t node, stored under the donor or receiver zone node depending of the storage. aR must contain information about distances and normals to bodies, defined by 'TurbulentDistance', 'gradxTurbulentDistance', 'gradyTurbulentDistance' and 'gradzTurbulentDistance', located at nodes or cell centers. Corrected points are defined in aR as points with cellN=2, located at nodes or cell centers. Parameter he is a constant, meaning that the interpolated points are pushed away of a distance he from the IBC points if these are external to the bodies. Parameter hi is a constant. If hi=0., then the interpolated points are mirror points of IBC points. If hi<0., then these mirror points are then pushed away of hi from their initial position. hi and he can be defined as a field (located at nodes or centers) for any point. If bodies are specified by tb, then the mirror points are automatically computed:

```
a = X.setIBCDData(aR, aD, order=2, penalty=0, nature=0, method='lagrangian', loc='nodes', storage='direct', he=0., hi=0. tb = None, cellNIBC='cellN', dim=3)
```

(See: Examples/Connector/setIBCDDataPT.py)

The following functions setInterpolations and chimeraTransfer are dedicated to elsA computations.

X.setInterpolations*: set the Chimera connectivity (EX points and cell centers to be interpolated, index for donor interpolation cell and interpolation coefficients).

Double wall projection technique is activated if 'double_wall=1'. Parameter 'sameBase=1' means that donor zones can be found in the same base as the current zone.

loc='cell' or 'face' indicates the location of the interpolated points (cell center or face). Interpolations with location at 'face' correspond to interpolation for EX points according to elsA solver.

prefixFile is the prefix for the name of the connectivity files generated for elsA solver (solver='elsA') or Cassiopee solver (solver='Cassiopee'). If prefixFile is not defined by user, no files are generated. nGhostCells is the number of ghost cells that are required by elsA solver when writing connectivity files.

If storage='direct', interpolation data are stored on the interpolated zone, if storage='inverse', interpolation data are stored in the donor zone as a ZoneSubRegion.t node:

```
t = X.setInterpolations(t, loc='cell', double_wall=0, storage='direct', prefixFile = "", sameBase=0, solver='elsA', nGhostCells=2)
```

(See: Examples/Connector/setInterpolationsPT.py)

X.chimeraTransfer: compute Chimera transfers. This function is compliant with the storage as it is defined for setInterpolations function. Parameter storage='direct' means that interpolation data are stored in interpolated zones, and parameter storage='inverse' means that interpolation data are stored in donor zones. Parameter 'variables' specifies the variables for which the transfer is applied.


```
t = X.chimeraTransfer(t, storage='direct', variables=[])
```

(See: Examples/Connector/chimeraTransferPT.py)

X.chimeraInfo: set information on Chimera connectivity, i.e. interpolated, extrapolated or orphan cells, donor aspect ratio and ratio between volume of donor and receiver cells. This function is compliant with the storage as it is defined for setInterpolations function. If type='interpolated', variable 'centers:interpolated' is created and is equal to 1 for interpolated and extrapolated cells, 0 otherwise. If type='extrapolated', variable 'centers:extrapolated' is created and its value is the sum of the absolute values of coefficients, 0 otherwise. If type='orphan', variable 'centers:orphan' is created and is equal to 1 for orphan cells, 0 otherwise. If type='cellRatio', variable 'centers:cellRatio' is created and is equal to $\max(\text{volD}/\text{volR}, \text{volR}/\text{volD})$ for interpolated and extrapolated cells (volR and volD are volume of receiver and donor cells). If type='donorAspect', variable 'centers:donorAspect' is created and is equal to the ratio between the maximum and minimum length of donor cells, and 0 for cells that are not interpolated:

```
t = X.chimeraInfo(t, type='interpolated')
```

(See: Examples/Connector/chimeraInfoPT.py)

1.4 Example files

Example file : Examples/Connector/connectMatch.py

```
# - connectMatch (array) -
import Generator as G
import Connector as X
import Geom as D
import Transform as T
import Converter as C
# 3D raccord i = 1 partiel profil NACA
msh = D.naca(12., 5001)
msh2 = D.line((1.,0.,0.),(2.,0.,0.),5001); msh = T.join(msh, msh2)
msh2 = D.line((2.,0.,0.),(1.,0.,0.),5001); msh = T.join(msh2, msh)
Ni = 300 ; Nj = 50
distrib = G.cart((0,0,0), (1./(Ni-1), 0.5/(Nj-1),1), (Ni,Nj,1))
naca = G.hyper2D(msh, distrib, "C")
res = X.connectMatch(naca,naca,sameZone=1,dim=2)
C.convertArrays2File([naca],"out.plt")
print res
```

Example file : Examples/Connector/blankCells.py

```
# - blankCells (array) -
import Converter as C
import Connector as X
import Generator as G
import Geom as D
import Transform as T

surf = D.sphere((0,0,0), 0.5, 20)
surf = C.convertArray2Tetra(surf)

a = G.cart((-1.,-1.,-1.),(0.1,0.1,0.1), (20,20,20))
ca = C.array('cellN',19,19,19)
ca = C.initVars(ca, 'cellN', 1.)
celln = X.blankCells([a], [ca], [surf], blankingType=1, delta=0.)
a = C.node2Center(a)
celln = C.addVars([a], celln)
C.convertArrays2File(celln, 'out.plt')
```

Example file : Examples/Connector/setHoleInterpolatedPts.py

```
# - setHoleInterpolatedPts (array) -
import Converter as C
import Connector as X
import Generator as G

def sphere(x,y,z):
    if x*x+y*y+z*z < 0.5**2 : return 0.
    else: return 1.

a = G.cart((-1.,-1.,-1.),(0.1,0.1,0.1), (20,20,20))
celln = C.node2Center(a)
celln = C.initVars(celln,'cellN', sphere, ['x','y','z'])
celln = X.setHoleInterpolatedPoints(celln,depth=1)
C.convertArrays2File([celln], 'out.plt')
```

Example file : Examples/Connector/optimizeOverlap.py

```
# - optimizeOverlap (array) -
import Converter as C
import Generator as G
import Transform as T
import Connector as X

Ni = 50; Nj = 50; Nk = 2
a = G.cart((0,0,0),(1./(Ni-1), 1./(Nj-1),1), (Ni,Nj,Nk))
b = G.cart((0,0,0),(2./(Ni-1), 2./(Nj-1),1), (Ni,Nj,Nk))
a = T.rotate(a, (0,0,0), (0,0,1), 10.)
a = T.translate(a, (0.5,0.5,0))

ca = C.node2Center(a); ca = C.initVars(ca, 'cellN', 1.)
cb = C.node2Center(b); cb = C.initVars(cb, 'cellN', 1.)
res = X.optimizeOverlap(a, ca, b, cb)
C.convertArrays2File(res, "out.plt")
```

Example file : Examples/Connector/maximizeBlankedCells.py

```
# - maximizeBlankedCells (array) -
import Converter as C
import Connector as X
import Generator as G

def F(x,y):
    if (x+y<1): return 1
    else: return 2

Ni = 50; Nj = 50
a = G.cart((0,0,0),(1./(Ni-1),1./(Nj-1),1),(Ni,Nj,1))
a = C.initVars(a, 'cellN', F, ['x','y'])
a = X.maximizeBlankedCells(a, 2)
C.convertArrays2File([a], 'out.plt')
```

Example file : Examples/Connector/setDoublyDefinedBC.py

```
# - setDoublyDefinedBC (array) -
import Converter as C
import Connector as X
import Generator as G

a = G.cart((0,0,0),(1,1,1),(10,10,10))
b = G.cart((2.5,2.5,-2.5),(0.5,0.5,0.5),(10,10,30))
celln = C.array('cellN',a[2]-1,a[3]-1,a[4]-1)
celln = C.initVars(celln, 'cellN', 1)
indmax = celln[2]*celln[3]
for ind in xrange(indmax):
```

```

        celln[1][0][ind] = 2
cellnb = C.array('cellN',b[2]-1,b[3]-1,b[4]-1)
cellnb = C.initVars(cellnb, 'cellN', 1)
celln = X.setDoublyDefinedBC(a, celln, [b], [cellnb], [1,a[2],1,a[3],1,1], depth = 1)
ac = C.node2Center(a)
ac = C.addVars([ac,celln])
C.convertArrays2File([ac], 'out.plt')

```

Example file : Examples/Connector/blankIntersectingCells.py

```

# - blankIntersectingCells (array)
import Converter as C
import Generator as G
import Transform as T
import Connector as X
a1 = G.cart((0.,0.,0.), (1.,1.,1.), (11,11,11))
a2 = T.rotate(a1, (0.,0.,0.), (0.,0.,1.),10.)
a2 = T.translate(a2, (7.,5.,5.))
A = [a1,a2]
Ac = C.node2Center(A); Ac = C.initVars(Ac,'cellN',1.);
Ac = X.blankIntersectingCells(A, Ac, tol=1.e-10)
C.convertArrays2File(Ac,"out.plt")

```

Example file : Examples/Connector/setDegeneratedBCPT.py

```

# - setDegeneratedBC (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Connector.PyTree as X
a = G.cylinder((0,0,0), 0., 1., 360., 0., 1, (21,21,21))
t = C.newPyTree(['Base']); t[2][1][2] = [a]
t = X.setDegeneratedBC(t)
C.convertPyTree2File(t,"out.cgns")

```

Example file : Examples/Connector/connectMatchPT.py

```

# - connectMatch (pyTree) -
import Generator.PyTree as G
import Connector.PyTree as X
import Converter.PyTree as C
import Transform.PyTree as T

a1 = G.cart((0.,0.,0.), (0.1, 0.1, 0.1), (11, 21, 3)); a1[0] = 'cart1'
a2 = G.cart((1., 0.2, 0.), (0.1, 0.1, 0.1), (11, 21, 3)); a2[0] = 'cart2'
t = C.newPyTree(['Base']) ; t[2][1][2] += [a1, a2]
t = X.connectMatch(t)
C.convertPyTree2File(t, 'out.cgns')

```

Example file : Examples/Connector/connectMatchPeriodicPT.py

```

# - connectMatch (pyTree) -
import Generator.PyTree as G
import Connector.PyTree as X
import Converter.PyTree as C

a = G.cylinder((0.,0.,0.), 0.1, 1., 0., 90., 5., (11,11,11))
t = C.newPyTree(['Base']) ; t[2][1][2] += [a]
t = X.connectMatchPeriodic(t,rotationCenter=[0.,0.,0.],translation=[0.,0.,5.])
t = X.connectMatchPeriodic(t,rotationCenter=[0.,0.,0.],rotationAngle=[0.,0.,90.])
C.convertPyTree2File(t, 'out.cgns')

```

Example file : Examples/Connector/connectNearMatchPT.py

```
# - connectNearMatch (pyTree) -
import Generator.PyTree as G
import Connector.PyTree as X
import Converter.PyTree as C
import Transform.PyTree as T

a1 = G.cart((0.,0.,0.), (0.1, 0.1, 0.1), (11, 21, 3)); a1[0] = 'cart1'
a2 = G.cart((1., 0.2, 0.), (0.1, 0.1, 0.1), (11, 21, 3)); a2[0] = 'cart2'
a2 = T.oneovern(a2,(1,2,1))
t = C.newPyTree(['Base']) ; t[2][1][2] += [a1, a2]
t = X.connectNearMatch(t)
C.convertPyTree2File(t, 'out.cgns')
```

Example file : Examples/Connector/getCEBBIntersectingDomainsPT.py

```
# - getCEBBIntersectingDomains (pyTree) -
import Connector.PyTree as X
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Internal as Internal

a = G.cart((0.,0.,0.), (0.1,0.1,0.1), (10,10,10)); a[0] = 'cart1'
b = G.cart((0.5,0.,0.), (0.1,0.1,0.1), (10,10,10)); b[0] = 'cart2'
c = G.cart((0.75,0.,0.), (0.1,0.1,0.1), (10,10,10)); c[0] = 'cart3'

t = C.newPyTree(['Cart'])
t[2][1][2] += [a, b, c]
bases = Internal.getNodesFromType(t, 'CGNSBase_t')
base = bases[0]
doms = X.getCEBBIntersectingDomains(base, bases, 1) ; print doms
```

Example file : Examples/Connector/getCEBBTimeIntersectingDomainsPT.py

```
# - getCEBBIntersectingDomains (pyTree) -
import Connector.PyTree as X
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Internal as Internal
from math import cos, sin

# Coordonnees du centre de rotation dans le repere absolu
def centerAbs(t): return [t, 0, 0]

# Coordonnees du centre de la rotation dans le repere entraine
def centerRel(t): return [5, 5, 0]

# Matrice de rotation
def rot(t):
    omega = 30.
    m = [[cos(omega*t), -sin(omega*t), 0],
          [sin(omega*t), cos(omega*t), 0],
          [0, 0, 1]]
    return m

# Mouvement complet
def F(t): return (centerAbs(t), centerRel(t), rot(t))

a = G.cylinder((0.,0.,0.), 0.5, 1., 360., 0., 2., (50,50,3))
# --- CL
a = C.addBC2Zone(a, 'wall', 'BCWall', 'jmin')
# --- champ aux noeuds
t = C.newPyTree(['Cylindre']) ; t[2][1][2].append(a)
# --- Equation state
t[2][1] = C.addState(t[2][1], 'EquationDimension', 3)
b = G.cylinder((1.5,0.,0.), 0.5, 1., 360., 0., 4., (50,50,3))
```

```

# --- champ aux centres
# --- CL
b = C.addBC2Zone(b, 'wall', 'BCWall', 'jmin')
t[2][1][2].append(b) ; b[0]='cylinder2'
#
dt = 1.; Funcs = [F,[]]
b1 = G.cart((-2.,-2.,0.),(0.4,0.4,1),(11,11,4))
b2 = G.cart((-4.,-2.,0.),(0.4,0.4,1),(11,11,4))
b3 = G.cart((2.,-2.,0.),(0.4,0.4,1),(11,11,4))
b4 = G.cart((-2.,-2.,3.),(0.4,0.4,1),(11,11,4))
b5 = G.cart((-4.,-2.,3.),(0.4,0.4,1),(11,11,4))
b6 = G.cart((2.,-2.,3.),(0.4,0.4,1),(11,11,4))
#
t = C.addBase2PyTree(t, 'Cart'); t[2][2][2] = [b1,b2,b3,b4,b5,b6]
t = C.initVars(t, 'centers:cellN', 1.)
t = C.initVars(t, 'Density', 2.)
bases = Internal.getNodesFromType(t, 'CGNSBase_t'); base = bases[0]
doms = X.getCEBBTimeIntersectingDomains(base, F, bases, Funcs, 0, 6, dt, sameBase=1)
print doms

```

Example file : Examples/Connector/applyBCOverlapsPT.py

```

# - applyBCOverlaps (pyTree) -
import Converter.PyTree as C
import Connector.PyTree as X
import Generator.PyTree as G

a = G.cylinder((0,0,0), 1., 1.5, 360., 0., 1., (30,30,10))
a = C.addBC2Zone(a, 'overlap1', 'BCOverlap', 'jmin')
t = C.newPyTree(['Base']); t[2][1][2].append(a)
t = X.applyBCOverlaps(t)
C.convertPyTree2File(t, 'out.cgns')

```

Example file : Examples/Connector/blankCellsPT.py

```

# - blankCells (pyTree) -
import Converter.PyTree as C
import Connector.PyTree as X
import Generator.PyTree as G
import Geom.PyTree as D
import Transform.PyTree as T

surf = D.sphere((0,0,0), 0.5, 20)

a = G.cart((-1.,-1.,-1.),(0.1,0.1,0.1), (20,20,20))
t = C.newPyTree(['Cart']); t[2][1][2].append(a)
t = C.initVars(t, 'centers:cellN', 1.)

bodies = [[surf]]
# Matrice de masquage (arbre d'assemblage)
import numpy
BM = numpy.array([ [ 1] ] )

t = X.blankCells(t, bodies, BM, blankingType='cell_intersect', delta=0.)
C.convertPyTree2File(t, 'out.cgns')

```

Example file : Examples/Connector/setHoleInterpolatedPtsPT.py

```

# - setHoleInterpolatedPoints (pyTree) -
import Converter.PyTree as C
import Connector.PyTree as X
import Generator.PyTree as G

def sphere(x,y,z):
    if x*x+y*y+z*z < 0.5**2 : return 0.
    else: return 1.

```

```

a = G.cart((-1.,-1.,-1.),(0.1,0.1,0.1), (20,20,20))
t = C.newPyTree(['Cart']); t[2][1][2].append(a)
t = C.initVars(t,'cellN', sphere, ['CoordinateX','CoordinateY','CoordinateZ'])
t = X.setHoleInterpolatedPoints(t,depth=1)
C.convertPyTree2File(t, 'out.cgns')

```

Example file : Examples/Connector/optimizeOverlapPT.py

```

# - optimizeOverlap (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Transform.PyTree as T
import Connector.PyTree as X

Ni = 50; Nj = 50; Nk = 2
a = G.cart((0,0,0),(1./(Ni-1), 1./(Nj-1),1), (Ni,Nj,Nk))
b = G.cart((0,0,0),(2./(Ni-1), 2./(Nj-1),1), (Ni,Nj,Nk)); b[0] = 'cart2'
a = T.rotate(a, (0,0,0), (0,0,1), 10.)
a = T.translate(a, (0.5,0.5,0))
t = C.newPyTree(['Base1', 'Base2'])
t[2][1][2].append(a); t[2][2][2].append(b)
t = X.optimizeOverlap(t)
C.convertPyTree2File(t, "out.cgns")

```

Example file : Examples/Connector/maximizeBlankedCellsPT.py

```

# - maximizeBlankedCells (pyTree) -
import Converter.PyTree as C
import Connector.PyTree as X
import Generator.PyTree as G

def F(x,y):
    if (x+y<1): return 1
    else: return 2

Ni = 50; Nj = 50
a = G.cart((0,0,0),(1./(Ni-1),1./(Nj-1),1),(Ni,Nj,1))
a = C.initVars(a,'cellN', F,
               ['CoordinateX','CoordinateY'])
a = C.node2Center(a, 'cellN')
a = C.rmVars(a,'cellN')
t = C.newPyTree(['Base',2]); t[2][1][2].append(a)
t = X.maximizeBlankedCells(t, 2)
C.convertPyTree2File(t, 'out.cgns')

```

Example file : Examples/Connector/cellN2OversetHolesPT.py

```

# - cellN2OversetHoles (pyTree) -
import Converter.PyTree as C
import Connector.PyTree as X
import Generator.PyTree as G

a = G.cart((0,0,0),(1,1,1),(10,10,10))
a = C.initVars(a, 'centers:cellN', 0)
a = X.cellN2OversetHoles(a)
t = C.newPyTree(['Base']); t[2][1][2].append(a)
C.convertPyTree2File(t, 'out.cgns')

```

Example file : Examples/Connector/oversetHoles2File.py

```

# - cellN2OversetHoles (pyTree) -
# - Dumping the OversetHoles node to files -
import Converter.PyTree as C
import Connector.PyTree as X
import Generator.PyTree as G

```



```

import Converter.Internal as Internal
import Converter

a = G.cart((0,0,0),(1,1,1),(10,10,10))
b = G.cart((0.5,0.5,0.5),(1,1,1),(10,10,10))
t = C.newPyTree(['Base1','Base2'])
t[2][1][2].append(a); t[2][2][2].append(b)
t = C.addBC2Zone(t, 'overlap1', 'BCOverlap', 'imin')
t = C.initVars(t, 'centers:cellN', 0)
t = X.applyBCOverlaps(t)
t = X.cellN2OversetHoles(t)

zones = Internal.getNodesFromType(t, 'Zone_t')
for z in zones:
    ho = Internal.getNodesFromType(z, 'OversetHoles_t')
    if ( ho != [] ):
        h = ho[0][2][1][1]
        dim = Internal.getZoneDim(z)
        h = Internal.convertIJKArray2lDArray(h, dim[1]-1,dim[2]-1,dim[3]-1)
        array = ['cell_index', h, h.size, 1, 1]
        Converter.convertArrays2File([array], 'hole_'+z[0]+'v3d',
                                     'bin_v3d')

```

Example file : Examples/Connector/setDoublyDefinedBCPT.py

```

# - setDoublyDefinedBC (pyTree) -
import Converter.PyTree as C
import Connector.PyTree as X
import Generator.PyTree as G

a = G.cart((0,0,0),(1,1,1),(10,10,10))
b = G.cart((2.5,2.5,-2.5),(0.5,0.5,0.5),(10,10,30)); b[0] = 'fente'
a = C.addBC2Zone(a, 'overlap1', 'BCOverlap', 'kmin',[b],'doubly_defined')
t = C.newPyTree(['Base1','Base2'])
t[2][1][2].append(a); t[2][2][2].append(b)

t = C.initVars(t, 'centers:cellN', 1)
t = X.applyBCOverlaps(t)
t = X.setDoublyDefinedBC(t)
C.convertPyTree2File(t, 'out.cgns')

```

Example file : Examples/Connector/blankIntersectingCellsPT.py

```

# - blankIntersectingCells (pyTree)
import Converter.PyTree as C
import Generator.PyTree as G
import Transform.PyTree as T
import Connector.PyTree as X

a1 = G.cart((0.,0.,0.),(1.,1.,1.),(11,11,11));
a2 = T.rotate(a1, (0.,0.,0.), (0.,0.,1.),10.)
a2 = T.translate(a2, (7.,5.,5.)); a1[0] = 'cart1'; a2[0] = 'cart2'
t = C.newPyTree(['Base']); t[2][1][2].append(a1); t[2][1][2].append(a2)
t = C.initVars(t,'centers:cellN',1.)
t2 = X.blankIntersectingCells(t, tol=1.e-10)
C.convertPyTree2File(t2, "out.cgns")

```

Example file : Examples/Connector/setInterpDataPT.py

```

# - setInterpolation (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Connector.PyTree as X

a = G.cylinder((0,0,0),1.,3.,360,0,1,(200,30,2)); a[0] = 'cylindre1'
a = C.addBC2Zone(a, 'wall1', 'BCWall', 'jmin')

```

```

a = C.addBC2Zone(a, 'ovl', 'BCOverlap', 'jmax')
b = G.cylinder((4,0,0),1.,3.,360,0,1,(200,30,2)); b[0] = 'cylindre2'
b = C.addBC2Zone(b, 'wall1', 'BCWall', 'jmin')
b = C.addBC2Zone(b, 'ovl', 'BCOverlap', 'jmax')
c = G.cart((-5.,-7.5,0), (15./200,15./200,1), (200,200,2))
t = C.newPyTree(['Corps1', 'Corps2', 'Bgnd'])
t[2][1][2].append(a); t[2][2][2].append(b); t[2][3][2].append(c)
t = X.connectMatch(t, dim=2)
t = C.fillEmptyBCWith(t, 'nref', 'BCFarfield', dim=2)
t = X.applyBCOverlaps(t, depth=1)
t[2][2:] = X.setInterpData(t[2][1], t[2][2:], loc='centers', storage='inverse')
C.convertPyTree2File(t, "out.cgns")

```

Example file : Examples/Connector/setInterpTransfersPT.py

```

# - setInterpTransfers (pyTree)-
import Converter.PyTree as C
import Connector.PyTree as X
import Generator.PyTree as G

a = G.cylinder( (0,0,0), 1, 2, 0, 360, 1, (60, 20, 3) )
b = G.cylinder( (0,0,0), 1, 2, 3, 160, 1, (30, 10, 3) )
a = C.addBC2Zone(a, 'wall', 'BCWall', 'jmin')
a = C.addBC2Zone(a, 'match', 'BCMatch', 'imin', a, 'imax', trirac=[1,2,3])
a = C.addBC2Zone(a, 'match', 'BCMatch', 'imax', a, 'imin', trirac=[1,2,3])
b = C.addBC2Zone(b, 'wall', 'BCWall', 'jmin')
b = C.addBC2Zone(b, 'overlap', 'BCOverlap', 'imin')
b = C.addBC2Zone(b, 'overlap', 'BCOverlap', 'imax')
t1 = C.newPyTree(['Base']); t1[2][1][2] = [a];
t2 = C.newPyTree(['Base']); t2[2][1][2] = [b]
t1 = C.fillEmptyBCWith(t1, 'nref', 'BCFarfield')
t2 = C.fillEmptyBCWith(t2, 'nref', 'BCFarfield')

t1 = C.initVars(t1, '{Density}= 1.')
t2 = C.initVars(t2, '{Density}=-1.')
t2 = C.node2Center(t2, ['Density'])

t2 = X.applyBCOverlaps(t2, depth=1)
t1 = X.setInterpData(t2, t1, double_wall=1, loc='centers',
                    storage='inverse', order=3)
t2 = X.setInterpTransfers(t2, t1, variables=['Density'])
C.convertPyTree2File(t2, 'out.cgns')

```

Example file : Examples/Connector/getOversetInfoPT.py

```

# - getOversetInfo (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Connector.PyTree as X

a = G.cylinder((0,0,0),1.,3.,360,0,1,(200,30,4)); a[0] = 'cylindre1'
a = C.addBC2Zone(a, 'wall1', 'BCWall', 'jmin')
a = C.addBC2Zone(a, 'ovl', 'BCOverlap', 'jmax')
b = G.cylinder((4,0,0),1.,3.,360,0,1,(200,30,4)); b[0] = 'cylindre2'
b = C.addBC2Zone(b, 'wall1', 'BCWall', 'jmin')
b = C.addBC2Zone(b, 'ovl', 'BCOverlap', 'jmax')
c = G.cart((-5.,-7.5,-2), (15./200,15./200,1), (200,200,8))
t = C.newPyTree(['Corps1', 'Corps2', 'Bgnd'])
t[2][1][2].append(a); t[2][2][2].append(b); t[2][3][2].append(c)
t = X.connectMatch(t, dim=3)
t = C.fillEmptyBCWith(t, 'nref', 'BCFarfield', dim=3)
t = X.applyBCOverlaps(t, depth=1, loc='centers')
t[2][1][2] = X.setInterpData(t[2][1][2], t, loc='centers', storage='direct', sameName=1)
t = X.getOversetInfo(t, t, loc='centers', type='interpolated')
t = X.getOversetInfo(t, t, loc='centers', type='orphan')
C.convertPyTree2File(t, "out.cgns")

```

Example file : Examples/Connector/setIBCDataPT.py

```
# - setIBCData (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Connector.PyTree as X
import Geom.PyTree as D
import Post.PyTree as P
import numpy as N
import Dist2Walls.PyTree as DTW
import Transform.PyTree as T

a = G.cart((-1,-1,-1),(0.01,0.01,1),(201,201,3))
s = G.cylinder((0,0,-1), 0, 0.4, 360, 0, 4, (30,30,5))
s = C.convertArray2Tetra(s); s = T.join(s); s = P.exteriorFaces(s)
t = C.newPyTree(['Base']); t[2][1][2] = [a]
# Blanking
bodies = [[s]]
BM = N.array([[1]],N.int32)
t = X.blankCells(t,bodies,BM,blankingType='center_in')
t = X.setHoleInterpolatedPoints(t,depth=-2)
# Dist2Walls
t = DTW.distance2Walls(t,[s],type='ortho',loc='centers',signed=1)
t = C.center2Node(t,'centers:TurbulentDistance')
# Gradient de distance localise en centres => normales
t = P.computeGrad(t, 'TurbulentDistance')
t = X.setIBCData(t, t, loc='centers', storage='direct')
C.convertPyTree2File(t,"out.cgns")
```

Example file : Examples/Connector/setInterpolationsPT.py

```
# - setInterpolation (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Connector.PyTree as X

a = G.cylinder((0,0,0),1.,3.,360,0,1,(200,30,2)); a[0] = 'cylindrel'
a = C.addBC2Zone(a, 'wall1', 'BCWall', 'jmin')
a = C.addBC2Zone(a, 'ovl', 'BCOverlap', 'jmax')
b = G.cylinder((4,0,0),1.,3.,360,0,1,(200,30,2)); b[0] = 'cylindre2'
b = C.addBC2Zone(b, 'wall1', 'BCWall', 'jmin')
b = C.addBC2Zone(b, 'ovl', 'BCOverlap', 'jmax')
c = G.cart((-5.,-7.5,0), (15./200,15./200,1), (200,200,2))
t = C.newPyTree(['Corps1', 'Corps2', 'Bgrd'])
t[2][1][2].append(a); t[2][2][2].append(b); t[2][3][2].append(c)
t = X.connectMatch(t, dim=2)
t = C.fillEmptyBCWith(t,'nref','BCFarfield', dim=2)
t = X.applyBCOverlaps(t, depth=1)
t = X.setInterpolations(t, loc='cell')
C.convertPyTree2File(t,"out.cgns")
```

Example file : Examples/Connector/chimeraTransferPT.py

```
# - chimeraTransfer (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Connector.PyTree as X

a = G.cylinder((0,0,0),1.,3.,360,0,1,(200,30,3)); a[0] = 'cylindrel'
a = C.addBC2Zone(a, 'wall1', 'BCWall', 'jmin')
a = C.addBC2Zone(a, 'ovl', 'BCOverlap', 'jmax')
b = G.cylinder((4,0,0),1.,3.,360,0,1,(200,30,3)); b[0] = 'cylindre2'
b = C.addBC2Zone(b, 'wall1', 'BCWall', 'jmin')
b = C.addBC2Zone(b, 'ovl', 'BCOverlap', 'jmax')
c = G.cart((-5.,-7.5,0), (15./200,15./200,1), (200,200,3))
```

```

t = C.newPyTree(['Corps1', 'Corps2', 'Bgrd'])
t[2][1][2].append(a); t[2][2][2].append(b); t[2][3][2].append(c)
t = X.connectMatch(t, dim=2)
t = C.fillEmptyBCWith(t, 'nref', 'BCFarfield', dim=3)
t = X.applyBCOverlaps(t, depth=2)
t = X.setInterpolations(t, storage = 'direct')

t = C.addVars(t, 'centers:Density')
t = C.addVars(t, 'centers:MomentumX')
t = C.addVars(t, 'centers:MomentumY')
t = C.addVars(t, 'centers:MomentumZ')
t = C.addVars(t, 'centers:StagnationEnergy')
for i in xrange(len(t[2])):
    t[2][i] = C.initVars(t[2][i], 'centers:Density', float(i+1))
    t[2][i] = C.initVars(t[2][i], 'centers:MomentumX', float(i+1))
    t[2][i] = C.initVars(t[2][i], 'centers:MomentumY', float(i+1))
    t[2][i] = C.initVars(t[2][i], 'centers:MomentumZ', float(i+1))
    t[2][i] = C.initVars(t[2][i], 'centers:StagnationEnergy', float(i+1))
t = X.chimeraTransfer(t, storage='direct', variables=['centers:Density', 'centers:MomentumX', 'centers:MomentumY', 'centers:MomentumZ', 'centers:StagnationEnergy'])
C.convertPyTree2File(t, 'out.cgns')

```

Example file : Examples/Connector/chimeraInfoPT.py

```

# - chimeraInfo (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Connector.PyTree as X

a = G.cylinder((0,0,0),1.,3.,360,0,1,(200,30,4)); a[0] = 'cylindrel'
a = C.addBC2Zone(a, 'wall1', 'BCWall', 'jmin')
a = C.addBC2Zone(a, 'ovl', 'BCOverlap', 'jmax')
b = G.cylinder((4,0,0),1.,3.,360,0,1,(200,30,4)); b[0] = 'cylindre2'
b = C.addBC2Zone(b, 'wall1', 'BCWall', 'jmin')
b = C.addBC2Zone(b, 'ovl', 'BCOverlap', 'jmax')
c = G.cart((-5.,-7.5,-2), (15./200,15./200,1), (200,200,8))
t = C.newPyTree(['Corps1', 'Corps2', 'Bgrd'])
t[2][1][2].append(a); t[2][2][2].append(b); t[2][3][2].append(c)
t = X.connectMatch(t, dim=3)
t = C.fillEmptyBCWith(t, 'nref', 'BCFarfield', dim=3)
t = X.applyBCOverlaps(t, depth=1)
t = X.setInterpolations(t, loc='cell', double_wall=1, storage='direct')
t = X.chimeraInfo(t, type='interpolated')
C.convertPyTree2File(t, "out.cgns")

```